OPEN ACCESS

Conference Proceedings Paper - Sensors and Applications

International Electronic Conference on Sensors and Applications
1 - 16 June 2014

www.mdpi.com/journal/sensors

Processing of mobile Multi-Agent Systems with a Code-based Agent Platform in Material-integrated Distributed Sensor Networks

Stefan Bosse[1*]

[1]University of Bremen, Dept. of Mathematics & Computer Science, Robert Hooke Str. 5, 28359 Bremen, Germany
[*] Author to whom correspondence should be addressed; E-Mail: sbosse@uni-bremen.de

Abstract: Multi-agent systems (MAS) can be used for a decentralized and self-organizing approach of data processing in a distributed system like a sensor network, enabling information extraction, for example, based on pattern recognition, decomposing complex tasks in simpler cooperative agents. MAS-based data processing approaches can aid the material-integration of Structural-Health-Monitoring applications, with agent processing platforms scaled to microchip level. A behaviour model suitable for distributed sensor network operations bases on an activity-transition graph (ATG) and is implemented in this work with program code holding the control and data state of an agent, which can be modified by the agent itself using code morphing techniques, and which is capable to migrate in the network between nodes. The program code is a self contained unit (a container) and embeds the agent data, the initialization instructions, and the ATG implementation. The microchip agent processing platform used for the execution of the agent code is a pipelined multi-stack virtual machine with a zero-operand instruction format, leading to small sized agent program code, low system complexity, and high system performance.

Keywords: Senor Networks; Multi-Agent System; Distributed Computing

1. Introduction

Structural Health Monitoring (SHM) of mechanical structures allows to derive not just loads, but also their effects to the structure, its safety, and its functioning from sensor data. A load monitoring system (LM) can be considered as a sub-class of SHM, which provides spatial resolved information about loads (forces, moments, etc.) applied to a technical structure.

Multi-agent systems (MAS) can be used for a decentralized and self-organizing approach of data processing in a distributed system like a sensor network, enabling information extraction, for example, based on pattern recognition **[5]**, decomposing complex tasks in simpler cooperative agents. MAS-

based data processing approaches can aid the material-integration of Structural-Health-Monitoring applications, with agent processing platforms scaled to microchip level which offer material-integrated real-time sensor processing. Agent mobility crossing different execution platforms in mesh-like networks and agent interaction by using tuple-space databases and global signal propagation aid solving data distribution and synchronization issues in the design of distributed sensor networks.

In **[3]**, the agent-based architecture considers sensors as devices used by an upper layer of controller agents. Agents are organized according to roles related to the different aspects to integrate, mainly sensor management, communication and data processing. This organization isolates largely and decouples the data management from the changing network, while encouraging reuse of solutions.

Usually sensor networks are a part of and connected to a larger heterogeneous computational network **[3]**. Employing of agents can overcome interface barriers arising between platforms differing considerably in computational and communication capabilities. That's why agent specification models and languages must be independent of the underlying run-time platform. Adaptive and learning behaviour of MAS, central to the agent model, can aid to overcome technical unreliability and limitations **[4]**.

This work bases on earlier data processing architectures described in **[2]** using virtual machines (VM) and mobile program code which can migrate between different VMs and nodes of a distributed (sensor) network. A code morphing mechanism was used to enable self modification of program code at run-time. This early approach matches only partially the agent model and had limited practically use due to very fine-grained code modification on instruction word level. Furthermore the VM architecture supported only coarse grained parallelism.

What is novel compared to other approaches?

- *Reliability* and *reactivity* provided by the *autonomy* of mobile state-based agents and *reconfiguration*.
- Agent mobility and *interaction* by using tuple-space databases and global signal propagation aid solving data distribution and synchronization issues in distributed systems design, and tuple spaces represent agent belief.
- One common agent *programming language* AAPL and *processing architecture* enables the synthesis of standalone parallel hardware implementations, alternatively standalone software implementations, and behavioural simulation models, enabling the design and test of large-scale heterogeneous systems.
- AAPL provides powerful statements for computation and agent control with static resources.
- A token-based pipelined multi-VM agent processing architecture suitable for hardware platforms with Register-Transfer Logic offering optimized computational resources and speed.
- Improved scaling in large network applications compared with full or semi centralized and pure message based processing architectures.

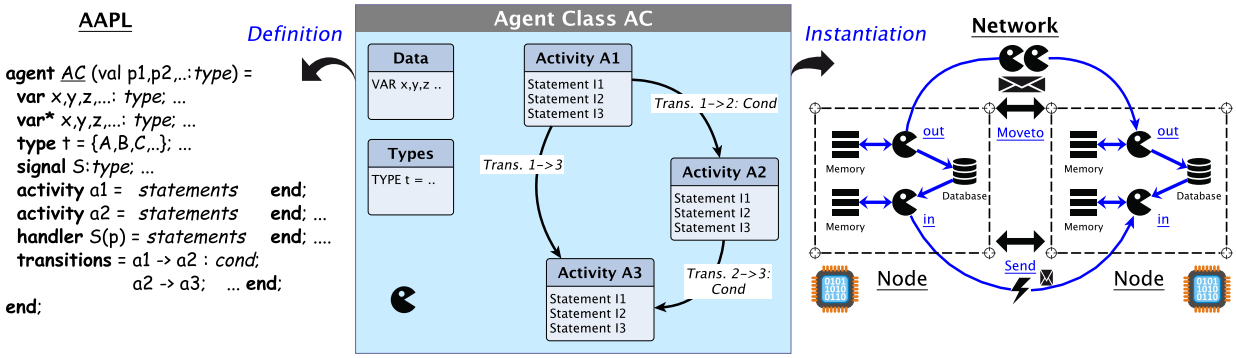
2. Modelling: The Activity-based Agent Model

The implementation of mobile multi-agent systems for resource constrained embedded systems with a particular focus on microchip level is a complex design challenge. High-level agent programming and behaviour modelling languages can aid to solve this design issue. Activity-based agent models can aid to carry out multi-agent systems on hardware platforms.

The behaviour of an activity-based agent is characterized by an agent state, which is changed by activities. Activities perform perception, plan actions, and execute actions modifying the control and data

state of the agent. Activities and transitions between activities are represented by an activity-transition graph (ATG). The activity-based agent-orientated programming language AAPL (detailed description in **[1]**) was designed to offer modelling of the agent behaviour on programming level, defining activities with procedural statements and transitions between activities with conditional expressions (predicates). Though the imperative programming model is quite simple and closer to a traditional PL it can be used as a common source and intermediate representation for different agent processing platform implementations (hardware, software, simulation) by using a high-level synthesis approach.

Figure 1. Agent behaviour programming level with activities and transitions (AAPL, left); agent class model and activity-transition graphs (middle); agent instantiation, processing, and agent interaction on the network node level (right) **[1]**.



The agent behaviour, perception, reasoning, and the action on the environment is encapsulated in agent classes, with activities representing the control state of the agent reasoning engine, and conditional transitions connecting and enabling activities. Activities provide a procedural agent processing by a sequential execution of imperative data processing and control statements. Agents can be instantiated from a specific class at run-time. A multi-agent system composed of different agent classes enables the factorization of an overall global task in sub-tasks, with the objective of decomposing the resolution of a large problem into agents in which they communicate and cooperate with one other.

The activity-graph based agent model is attractive due to the proximity to the finite-state machine model, which simplifies the hardware implementation.

An activity is activated by a transition depending on the evaluation of (private) agent data (conditional transition) related to a part of the agents belief in terms of BDI architectures, or using unconditional transitions (providing sequential composition), shown in Fig. **1**. Each agent belongs to a specific parameterizable agent class ***AC***, specifying local agent data (only visible for the agent itself), types, signals, activities, signal handlers, and transitions.

The *AAPL programming language* offers statements for parameterized agent instantiation, like the creation of new agents and the forking of child agents, using the `new(args)` and `fork(args)` statements, respectively. Furthermore, agent interaction by using synchronized Linda-like tuple database space access and signal propagation (messages carrying simple data delivered to asynchronous executed signal handlers), agent mobility (migration using the `moveto` statement), and statements for ATG transformations and composition. Transitions and activities can be added, removed, or changed at run-time with the $\texttt{transition}^{+/-/*}(A_i,A_j,\texttt{cond})$ and $\texttt{activity}^{+/-}(A_i)$ statements, respectively. Access of the tupel space is granted by using `in(TP)`, `rd(TP)`, `rm(TP)`, `exist?(TP)` and `out(T)` primitives (`T`: n-dimensional value tuple, `TP`: n-dimensional tuple with value patterns). This tuple-space approach can be used to build distributed data structures and the atomicity of tuple operations provides

data structure locking. A signal SIG can be sent to an (remote) agent with the identifier ID by using the `send(ID,SIG,ARG)` statement.

3. Architecture: Agent Processing Platform

The requirements for the agent processing platform is summarized: 1. to be suitable for microchip level (SoC) implementations, 2. supporting a standalone platform without any operating system, 3. performing efficient parallel processing of a large number of different agents, 4. to be scalable regarding to the number of agents processed concurrently, and 5. providing the capability to create, modify, and migrate agents at run-time. Migration of agents requires the transfer of the data and control state of the agent between different virtual machines (at different node locations). To simplify this operation, the agent behaviour based on the activity-transition graph model is implemented with program code, which embeds the (private) agent data as well as the activities, the transition network, and the current control state. It can be handled as a self contained execution unit. The execution of the program by a virtual machine (VM) is handled by a task. The program instruction set consists of zero-operand instructions, mainly operating on the stacks.

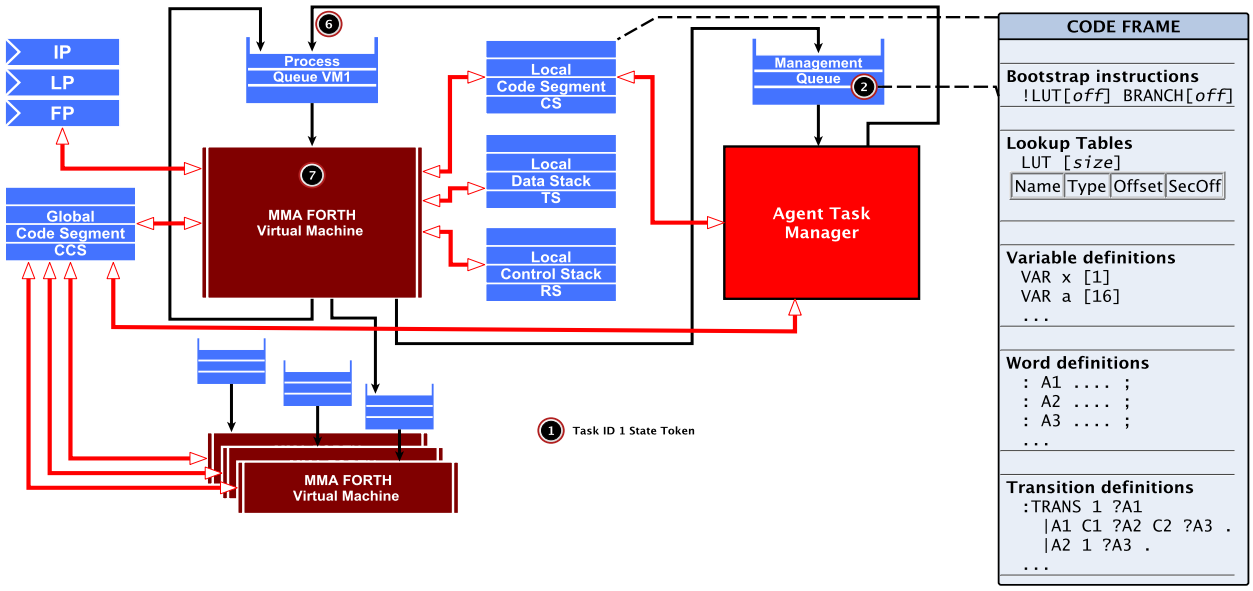
The virtual machine executing tasks is based on a traditional FORTH architecture based on an extended zero-operand word instruction set (αFORTH): a data (*TS*) and a control (*RS*, return) stack, a code segment (*CS*) storing the program code with embedded data, shown in Fig. **2**. The program is mainly organized by a composition of words (functions). A word is executed by transferring the program control to the entry point in the *CS*; arguments and computation results are passed only by the stack(s). There are several virtual machines with each attached to (private) stack and code segments. There is one global code segment (*CCS*) storing global available functions which can be accessed by all programs. This multi-segment architecture ensures high-speed program execution and. the local *CS* can be implemented with (asynchronous) dual-port RAM (the other side is accessed by the agent manager, discussed below), the stacks with simple single-port RAM. The global *CCS* requires a Mutex scheduler to resolve competition by different VMs.

Commonly the number of agent tasks N_A executed on a node is much larger than the number of available virtual machines N_V. Thus efficient and well balanced multi-task scheduling is required to get proper response times of individual agents. To provide fine grained granularity of task scheduling, a token based pipelined task processing architecture was chosen. A task of an agent program is assigned to a token holding the task identifier of the agent program to be executed. The token is stored in a queue and consumed by the virtual machine from the queue. After a (top-level) word was executed, leaving an empty data and return stack, the token is either passed back to the processing queue or to another queue (processing queues of different VMs or the agent manager). This task scheduling policy allows fair and low-latency multi-agent processing with fine grained scheduling.

The program code frame (shown on the right of Fig.**2**) of an agent consists basically of four parts: 1. embedded agent body variable definitions and lookup tables, 2. word definitions defining agent activities (procedures without arguments and return values) and generic functions, 3. bootstrap instructions which are responsible to setup the agent in a new environment (e.g, after migration or at first run), and 4. the transition network calling activity words (defined above) and branching to the next activity execution depending on the evaluation of conditional computations with private data (variables). The transition network section can be modified by the agent by using special instructions. Furthermore, new agents can be created by composing activities and transitions from existing agent programs, creating

sub-classes of agent super classes with a reduced functionality. The program frame (referenced by the frame pointer *FP*) is stored in the local code segment of the VM executing the program task (using the instructions pointer *IP*). The initial code frame loading and any modifications of the code are performed by the agent task manager only. A migration of the program code between different VMs requires a copy operation. Each time a program task is executed and after returning from the current activity execution the stacks are assumed to be empty. Each VM has only one stack shared by all program tasks executed on the VM! This design significantly reduces the required hardware resources. Therefore, the return from an agent activity word execution is an appropriate task scheduling point for a different task transferred from the VM processing queue.

Figure 2. The Agent processing architecture based on a pipelined stack-based Virtual Machine approach. Tasks are executing units of agent code, which are assigned to a token passed to the VM by using processing queues. A task points to the next instruction word to be executed. After execution the task token is either passed back to the input processing queue or to another queue of either the agent manager or a different VM. Right: the content and format of a code frame.



4. Methods: Agent Behaviour Programming and Modification with Code Morphing

The αFORTH instruction set consists of a generic FORTH sub-set I_D with common data processing words operating on the data and return stack used for computation and a special instruction set I_P for agent related processing and agent behaviour modification at run-time. Take a look at the following very simple αFORTH code Ex. **1** implementing an agent performing a mean value calculation of sensor values exceeding a threshold (agent parameter `thr`) with two body variables `x` and `m`, three activities $\{A_1, A_2, A_3\}$, and a transition network with some conditional transitions. The AAPL behaviour model is shown on the right side. The sensor value is retrieved from and finally passed to the tuple database.

An αFORTH program frame (see Fig. **2**) starts with the program lookup relocation table (*LUT*, line 2), preceded by bootstrap instructions setting the *LUT* offset register *LP* and the program counter *IP* pointing to next instruction to be executed. This *LUT* is a reserved area in the program frame initially empty and will not be transferred on migration, and is used by the VM. A *LUT* row consists of the entries: {*Name, Type, Code Offset, Secondary Offset*}. Within the program code, address references

of variables, words, and transitions are relocated by the *LUT*. This indirect address access mechanism allows simplified reconfiguration of the program at run-time by the agent task manager. If a program frame is executed the first time the *LUT* is updated and filled with entries. A variable `VAR V` and a word definition `:W` creates, a transition definitions `|W` updates an entry in the *LUT*.

Example 1. Left code shows an αFORTH program derived from an agent behaviour specification in AAPL on the right side.

```
αFORTH                                ⇔  AAPL
1  !LUT(4) BRANCH(4)
2  LUT[32]                                agent mean_filter(thr:int) =
3  VAR x[INT] VAR m[INT]                    var x,m: integer;
4  :A1 "SENSOR0" ? IN2 x ! ;                activity A1 = in(SENSOR0,x?); end;
5  :A2 m @ x @ + 2 / m ! ;                  activity A2 = m:=(m + x)/2; end;
6  :A3 "SENSOR0" m OUT2 KILL ;              activity A3 = out(SENSOR0,m); self#kill; end;
7  :TRANS 1 ?A1                             transitions =
8    |A1 x thr > ?A2 x thr <= ?A3 .           A1→A2:x>thres; A1→A3:x<=thres; A2→A3;
9    |A2 1 ?A3 .                            end;
10 ;                                    end;
```

The `?W` statement branches to the transition sequence of `W` (starting with `|W`, a row in the transition table terminated by a dot word) if the top stack element (from a previous boolean computation) is true. The first time this statement is called the corresponding `|W` is searched in the `:TRANS` section and the code offset is stored in the lookup table. Further calls can be resolved by the *LUT*. The `|W` statement calls the word `W`. After the return the transition selection sequence is executed selecting the next activity word to be executed.

Beside pure procedural activity words (w/o any data passing leaving the data stack unchanged) there are functional words passing arguments by using the data stack. These words can be exported (`EXPORT W`) to a global dictionary (transferring the code to a *CCS* frame) and reused by other agents which can import these functions (`IMPORT W`), which creates a *LUT* entry pointing to the *CCS* code frame and offset relative to this frame. Global functions may not access any private agent data.

Example 2. Code morphing and agent creation related with agent behaviour modification.

```
αFORTH                                ⇔  AAPL
1  SELF TRAN* |A1 x y < ?A2 .             transition*(A1,A2,x < y);  -- replace all transitions A1->A2
2  SELF TRAN+ |A1 x 0 = ?A3 .             transition+(A1,A2,x = 0);  -- add transition A1->A2
3  NEW DUP ACT+ A1 A2 A5 .                a := new();            -- create empty agent
                                          a#activity+(A1,A2,A5); -- add activities to new empty agent
   DUP TRAN+ |A1 1 ?A2 .                  a#transition+(A1,A2);  -- add transition(s)
   100 FORK                               a#fork(100);           -- create and start agent with argument
4  SELF 100 FORK                          a := fork(100); -- fork child agent with different argument(s)
```

Reconfiguration of the ATG modifying the agent behaviour using code morphing (see Ex. **2**) enables agent sub-classing at run-time. E.g., used in the employment of parent-child systems creating child agents having a operationally reduced sub-set from the parent agent. This approach has the advantage of high efficiency and performance due to reduced code size. New agents can be created by simply forking an existing agent (`FORK`), which creates a copy of the parent agent including the data space. New agent programs (with different behaviour) can be created by composing existing activities and by creating new transitions (`NEW`). The capability to change an existing agent is limited to the modification of the transitions and by removing activities. Modification of transitions can invalidate *LUT* entries which are updated on the fly. The transition table modification (and activity deletion) is the main tool for run-time

adaptation of agents based on learning. The modified agent behaviour can be inherited by forked child agents.

5. Discussion and Conclusions

In this work, a novel **sensor data processing approach** using mobile agents for reliable distributed and parallel data processing in large scale networks of low-resource nodes was introduced, leading to a sensor signal pre-processing at run-time inside the sensor network, which reduces communication significantly. This mobile program-based agent approach is well suited for massive distributed multi-agent systems with a common cooperate goal. Examples are self-organizing systems used for pattern and feature recognition **[1]** or event-based sensor distribution in large-scale networks. These distributed algorithms require replication and diffusion behaviours with neighbourhood exploration by forked child agents delivering pre-computed information parts (divide and conquer strategy). Agent mobility crossing different execution platforms in mesh-like networks and agent interaction by using tuple-space databases and global signal propagation aid solving data distribution and synchronization issues in the design of distributed heterogeneous sensor networks. The program code can be efficiently compiled form a high-level agent behaviour specification using the AAPL programming language. The agent processing VM was adapted to and optimized for AAPL specific statements and behaviours. A typical program code size of an agent employed in sensor networks for sensor pre-processing and distribution is about 1000 words (assuming a 16 bit machine requiring only 2000 Bytes). Migration of agents require only the transfer of the program code encapsulated in messages. A migrated program code frame can be started immediately on the new node or VM leading to short start-up times.

6. References

1. S. Bosse, *Distributed Agent-based Computing in Material-Embedded Sensor Network Systems with the Agent-on-Chip Architecture*, IEEE Sensors Journal, Special Issue on Mateiral-integrated Sensing, DOI 10.1109/JSEN.2014.2301938
2. S. Bosse, F. Pantke, *Distributed computing and reliable communication in sensor networks using multi-agent system*, Production Engineering, Research and Development, 2012, ISSN: 0944-6524, DOI:10.1007/s11740-012-0420-8
3. M. Guijarro, R. Fuentes-fernández, and G. Pajares, *A Multi-Agent System Architecture for Sensor Networks*, Multi-Agent Systems - Modeling, Control, Programming, Simulations and Applications, 2008
4. C. Sansores, J. Pavón, *An Adaptive Agent Model for Self-Organizing MAS*, in Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008), May, 12-16., 2008, Estoril, Portugal, 2008, pp. 1639–1642.
5. X. Zhao, S. Yuan, Z. Yu, W. Ye, J. Cao. (2008), *Designing strategy for multi-agent system based large structural health monitoring*, Expert Systems with Applications, 34(2), 1154–1168. doi:10.1016/j.eswa.2006.12.022